LA-UR- 02-6058

Approved for public release; distribution is unlimited.

Title:

A Compact Machine Representation Language for Simulation of Large-scale Parallel Architectures

Author(s):

Nick Moss

Submitted to:

world wide web

A Compact Machine Representation Language for Simulation of Large-scale Parallel Architectures*

Nick Moss - nickm@lanl.gov
Modeling, Algorithms, and Informatics Group
Los Alamos National Laboratory

November 4, 2002

Abstract

The Parallel Architecture Simulation Tool (ParSim)[1], built upon the discrete event simulation framework DaSSF (Darthmouth Scalable Simulation Framework) allows detailed simulation and predicitive evaluation of a wide range of parallel architectures. Representational flexibility is achieved with the Domain Modeling Language (DML), through which DaSSF separates model configuration and initialization from implementation. Nodes, network interface cards, and switches, are implemented in C++ but instantiated, parameterized and interconnected in DML; this code or "model.dml" constitutes a complete ParSim model. This approach provides flexibility, but manipulation at this level is difficult when working with large models, where components and their interconnections are on the order of thousands. Previously, specialized scripts were developed to assemble and interconnect components and output DML for particular architectures. The model description parser presented in this paper essentially follows the same procedure, but automates and provides a generalized interface for common elements of the process. The model description is a domain-specific DML translator allowing high-level manipulation of ParSim models and improved handling in their creation, modification, and readability.

1 Introduction

ParSim is an ongoing project to produce high fidelity simulations[2] of ASCI/Extreme-scale (more than one thousand processors) parallel architectures with an end goal of providing predictive capability in the evaluation of existing and hypothetical architectures, and the performance of programs run on them. We model machines as constructed from a set of basic components including switches, nodes, and network interface cards. The components approximate their real counterparts in varying levels of detail and include parameters for bandwidth, transmission delay, and others in the case of switches; for nodes and network interface cards possible parameters include packet size, routing methods, and others which determine messaging behavior. The underlying network is modeled at the packet level and based on the Quadrics Elan 3 protocol. Message and packet transmission through the

*This work was funded by the U.S. Department of Energy under ASCI Discom2.

simulated network, and the trace events generated through this process, provide useful data for analyzing performance. Another approach, using ParSim's direct execution feature, allows any MPI program to execute on the simulated machine as if it were executed on the actual machine. Because performance is highly dependent on applications and how well they run in a parallel environment, direct execution is a powerful method of analysis in the evaluation of hypothetical architectures.

1.1 Framework and Implementation

ParSim is built on Darthmouth Scalable Simulation Framework[4], a C++ implementation of the Scalable Simulation Framework specification (SSF). DaSSF is well suited for discrete event simulations of large scale such as those we are interested in studying. The SSF specification originally envisioned simulating the Internet and other massive dynamic networks, and for ParSim, where the network is a central component, the SSF API is a natural fit. ParSim component classes define network components, routing methods, messages, packets and other constructs. They are implemented in C++ and derived from the base classes defined by the SSF specification: *Entity*, *Event*, *Process*, *Input Channel*, and *Output Channel*.

1.2 DML

A model may be constructed by instantiating components within C++ code, but DaSSF provides a better approach using the Domain Modeling Language (DML), which isolates model specification from implementation. DML is a generic syntax which supports recursive-like nesting of submodels, or subsections of DML code, and relatively convenient specification of large structures. DaSSF adapts the generic DML syntax, adding various extensions and defining its semantics. DaSSF uses DML for specification of runtime parameters `runtime.dml`, runtime architectures `machine.dml`, and most significantly, model construction `model.dml`.

1.3 Machine Representation in DML

ParSim uses DML in model construction for the instantiation and parameterization of NICs, nodes, switches, and for specification of their interconnection. This trivial model shows how entities are declared and interconnected in DML:

```
 ENTITY [
    INSTANCEOF "Node"
    PARAMS [
      INT    0
    ]
]

 ENTITY [
    INSTANCEOF "NIC"
    PARAMS [
      INT    1
    ]
]
```

```
 ENTITY [
    INSTANCEOF "Switch"
    PARAMS [
      INT    2
    ]
]

MAP [FROM 0(NETOUT) TO 1(NETIN) DELAY 1]
MAP [FROM 1(NETOUT) TO 0(NETIN) DELAY 1]
MAP [FROM 1(NETOUT) TO 2(LINKIN0) DELAY 1]
MAP [FROM 2(LINKOUT0) TO 1(NETIN) DELAY 1]
```

where Node is the name of the C++ class we are casting from. The `PARAMS[]` section specifies the required parameters and additional optional parameters are set inside `CONFIGURE[]` (not shown).

Each `MAP` statement corresponds to channel mapping in one direction. So

```
MAP [FROM 0(NETOUT) TO 1(NETIN) DELAY 1]
```

establishes an outgoing connection from node ID 0 to NIC ID 1. In the types of architectures we are interested in, components are always wired in both directions, so the DML statement above is accompanied by

```
MAP [FROM 1(NETOUT) TO 0(NETIN) DELAY 1]
```

Finally, the following statements connect the NIC to port 0 on the switch:

```
MAP [FROM 1(NETOUT) TO 2(LINKIN0) DELAY 1]
MAP [FROM 2(LINKOUT0) TO 1(NETIN) DELAY 1]
```

1.4 Motivation

A complete ParSim model includes a collection of DML statements exemplified by the code above, with a section declaring and parameterizing each component, and MAP directives for each of their interconnections. For large architectures, despite DML's recursive features, the DML for large models can be quite lengthy (several thousands of lines) and difficult to manage. This document details the design and usage of a domain-specific language and parser for DML translation which allow for high-level manipulation and easier management of large-scale ParSim models.

2 Model Description Parser

The parser and associated files are organized as

```
parse
components/
models/
```

The `parse`[1] program is called as `parse` *`input_model_description`*. Sample model descriptions are included in `models`. `components` contains DML code for commonly used components. Detailed explanations of components, design and operation of the parser, and the usage of the model description language are included in the following sections.

2.1 Components

The parser views components as one of two basic types: switches or nodes. A switch embodies the DML code that instantiates a TSwitch entity and nodes include initialization code for both TSMPNode and TNIC (the C++ implementation class names for these components). The code associated with a switch typically looks like:

```
ENTITY [
    ID $ID
    INSTANCEOF "TSwitch"
    PARAMS [
      INT $ID
      INT 8
      INT 2
    ]
    CONFIGURE [
      PACKET_DELAY  $PACKET_DELAY{50}
      NET_BANDWIDTH $NET_BANDWIDTH{400}
    ]
  ]
```

While a node clusters TNIC and TSMPNode together as:

```
CLUSTER [
    ID $ID
    ENTITY [
    INSTANCEOF "TSMPNode"
    PARAMS [
      INT    $ID
      INT    $ID
      STRING "$ID"
    ]
    CONFIGURE [
      MESSAGES [
        $MESSAGES{}
        ]
    ]
  ]

ENTITY [
    INSTANCEOF "TNIC"
    PARAMS [
      INT    $ID1
      STRING "$ID"
      INT    $3{8}
    ]
```

[1]implemented in C using lex and yacc

```
      CONFIGURE [
        MESSAGE_DELAY $MESSAGE_DELAY{5}
        PACKET_DELAY  $PACKET_DELAY{1}
        PACKET_SIZE   $PACKET_SIZE{320}
        METHOD        "$METHOD{Quadrics1}"
        BUS_BANDWIDTH $BUS_BANDWIDTH{200}
        NET_BANDWIDTH $NET_BANDWIDTH{400}
        ACK_BYTES     $ACK_BYTES{64}
      ]
   ]
  MAP [FROM 0(BUSOUT) TO 1(BUSIN) DELAY $channel_delay{5}]
  MAP [FROM 1(BUSOUT) TO 0(BUSIN) DELAY $channel_delay{5}]
  ]
```

There are no restrictions on the type of code that may be associated with a switch or node, but this division allows for consistent ordering and ID assignment as it should appear in the output model DML and also simplifies component interconnectivity.

2.2 Component Templates

A component's code is included in the description file or read from a template residing in the `components` directory. A template may be associated with any component and typically includes DML code and additional variables, labeled as `$VARIABLE_NAME`. These are either automatically set by the parser (e.g `$ID` is substituted for the corresponding component ID) or by a statements within the model description file. Parameters should be included within the template as `$<parameter name>{<default value>}`, e.g: `$PACKET_SIZE{320}`. The code segments above are examples of a typical switch and node template and illustrate the usage of these two types of embedded variables.

The component type is determined by its associated template, where template name is identical to the component symbol as it appears in the model description file. Template location in `components/switch` or `components/node` determines type or is stated explicitly when declaring a template within the model description.

2.3 Output Structure

The unique file `components/main` defines default top-level structure of the output `model.dml`, providing the final glue for postproccesed DML segments. In most cases the structure is simple and it is sufficient to use the default template `components/main` which is supplied as:

```
parsim [
  node [
    ID 0
    INSTANCEOF "TSMPNode"
  ]
  nic [
    ID 1
    INSTANCEOF "TNIC"
  ]
  switch [
    INSTANCEOF "TSwitch"
  ]
]

MODEL [
$nodes
$switches
$connections
]
```

Here there are references to three variables set in parsing: `$nodes, $switches, $connections`. These variables are interpolated for the corresponding DML segments generated by the parser. Typically they are output as a continuous block but are kept separate in case any other code or comments are to be included between them. In addition to the output files specified in the model description, three files `switches.dml, nodes.dml`, and `connections.dml` are created which contain these as separate blocks of output.

2.4 ID Assignment

Components are assigned ID's according to type and order of appearance in layer assignment statements. Node IDs start at 0 with switches starting at 2∗*DML_ID_MULTIPLIER*. The C constant *DML_ID_MULTIPLIER* is currently set to 100000 so switch IDs start at 200000. A component's ID is referenced by inclusion of `$ID` within the template. Additionally, `$IDx` where $0 < x < 10$, outputs x∗*DML_ID_MULTIPLIER*+*ID*. This is used to specify the absolute ID for a NIC in:

```
    ENTITY [
        INSTANCEOF "TNIC"
        PARAMS [
          INT    $ID1
          STRING "$ID"
          ...
```

3 Model Description Language

The parser takes as input a description file that assembles DML and outputs to the desired files. Input to the parser consists of four basic types of statements, with their usage summarized below.

3.1 Parameter Assignment

Syntax: *component_symbol:component_property component_value*
Examples:

```
    n:PACKET_DELAY{10}

    n:MESSAGES{
     MESSAGE [
             TIME   0
             TARGET "1"
```

```
            SIZE   384
          ]
          MESSAGE [
            TIME   28000
            TARGET "1"
            SIZE   704
          ]
          MESSAGE [
            TIME   84000
            TARGET "1"
            SIZE   1024
          ]
          MESSAGE [
            TIME   168000
            TARGET "1"
            SIZE   1344
          ]
}
```

sets the node's PACKET_DELAY to 10 assuming `$PACKET_DELAY{<default value>}` was defined for the component `n`.

Component parameters are typically named with the convention that parameters directly corresponding to DML parameters are capitalized, numbered for parameters appearing in `PARAMS[]` section, or lower-cased otherwise. For example, a node template is defined as:

```
ENTITY [
    _extends .parsim.nic
    PARAMS [
      INT    $ID1
      STRING "$ID"
      INT    $3{8}
    ]
    CONFIGURE [
      MESSAGE_DELAY $MESSAGE_DELAY{5}
      PACKET_DELAY  $PACKET_DELAY{1}
      PACKET_SIZE   $PACKET_SIZE{320}
      METHOD        "$METHOD{Quadrics1}"
      BUS_BANDWIDTH $BUS_BANDWIDTH{200}
      NET_BANDWIDTH $NET_BANDWIDTH{400}
      ACK_BYTES     $ACK_BYTES{64}
    ]
  ]
 MAP [FROM 0(BUSOUT) TO 1(BUSIN) DELAY $channel_delay{5}]
 MAP [FROM 1(BUSOUT) TO 0(BUSIN) DELAY $channel_delay{5}]
 ]
```

and parser statements set various parameters with:

```
n:3{10} # sets third parameter to 10
n:channel_delay{4}
n:PACKET_DELAY{10}
```

3.2 Layer Declaration

Syntax: layer_name=layer_components
Examples:

```
# declare a layer with 64 nodes
l0=n[64]
# layer with 16 nodes
l1=s[16]
# individual components included in layer
layer=s,s1,s,s,a,s,s
```

3.2.1 Description

The symbols `l0` and `l1` are arbitrary and are used for reference at later points to specify connections. Additionally, any component can be named freely but numbers at the end of a symbol have special significance: they declare a distinct instance of their parent (parents are labeled without number or number=0, e.g: `n, n0`), copying and maintaining all properties from their parent until redefined.

3.3 Connect Directives

In both cases, layer to layer or component to component, connect statements set internal connections used to output the corresponding DML `MAP[]` statements.

3.3.1 Layer to Layer Wiring

Syntax: *connect source_layer,destination_layer <mapping vector> <delay vector>*
Examples:

```
connect l0,l1
connect l0,l1 [2,1,0]
connect l0,l1 [] [2,1]
connect 10,l1 [1,0,2,3,4,5] [3,3]
```

The first example (no mapping vector is provided) wires source to destination iterating through source components and connection port 0 of each source to the the first available port > 4 on the destination layer. IMPORTANT: This is the method to use in wiring nodes to the first layer and no other interface exists other than manual component to component wiring (next section). The reason for this is that nodes are not assigned a base-four ID and all other connect statements calculate their destination based on a source base-four ID.

The second example includes a mapping matrix `[2,1,0]` that maps a connection from a source component to a component in the destination layer as a function of the base four digits of the source ID and source port[3].

The third example doesn't use a mapping vector but includes an empty `[]` as a place-holder. The second vector specifies outgoing and incoming channel delay respectively.

3.3.2 Component to Component Wiring

Syntax: *source_component<port>:destination_component1<port>,destination_component2<port>,... <wire_vector> <delay_vector>*
Examples:

```
connect 1:2,3,4,5
connect 0:1(0),2(1),3(2),4(3)
connect 1(7):200000(0)
```

The first example wires a component with ID 1 to components corresponding to ID 2,3,4,5 in order of lowest available port with source ports restricted to 0 through 3 and destination ports 4 through 7.

The second example wires component ID 1's available ports to specific ports on the output ID's. Note: whenever a port is stated explicitly, the source and destination port restrictions stated above do not apply, it may map any port 0 through 7.

The third example wires a specific source port (7) on a component (ID 1) to a specific destination port (0 on component ID 200000).

For all connect statements, either layer to layer or component to component, attempting to remap a port triggers a warning, but the action allowed.

3.4 Output Blocks

3.4.1 File Output

Syntax: *file_name<source>*
Examples:

```
        runtime.dml{
        ...
        runtime DML
        ...
}
```

The example above takes anything included within {} and writes it to the specified output file, `runtime.dml`, interpolating values for `$nodes, $switches, $connections`.

The line below opens `components/main` (since source is empty) and writes the contents to model.dml, interpolating parser values in the same fashion as the previous example:

```
model.dml{}

{parsim [
  node [
    ID 0
    INSTANCEOF "TSMPNode"
  ]
  nic [
    ID 1
    INSTANCEOF "TNIC"
  ]
  switch [
    INSTANCEOF "TSwitch"
  ]
]

MODEL [
$nodes
$switches
$connections
]
}
```

The functionality in the example above is identical to the previous two but writes to *stdout* since *file_name* was not specified.

3.4.2 Template Output

Syntax:

```
type component_name{
...
DML code
...
}
```

Examples:

```
node n{CLUSTER [
    ID $ID
    ENTITY [
    _extends .parsim.node
    PARAMS [
      INT    $ID
      INT    $ID
      STRING "$ID"
    ]
...
}

switch s{ENTITY [
    ID $ID
    _extends .parsim.switch
    PARAMS [
      INT $ID
      INT 8
      INT 2
    ]
    CONFIGURE [
      PACKET_DELAY  $PACKET_DELAY{50}
      NET_BANDWIDTH $NET_BANDWIDTH{400}
    ]
  ]
}
```

The code above declares a component template for a node and switch. The template may include `$VARIABLES` and overall is treated the same as if it were read from the components directory, and can now be included in any layer declaration statements. Note: defining a template overrides any previous templates read from the components directory in cases where the name is not unique.

4 A Complete Example

A model of a 64 node cluster *wolverine* with three layers of switches is constructed in the model description below. Here, the typical component templates `components/node/n` and `components/switch/s` are used. Switch `s` uses its default parameters defined within the template. Node `n` also uses default values for some parameters, but redefines others in section 1. Parameter values may span multiple lines as seen in `n1:MESSAGES{...}`. This statement is different from the previous ones in that it sets up a new component template `n1` which inherits all values set for n up to this point and also sets the node's MESSAGES parameter to the specified value. This component can now be instantiated in layer declarations as in section 2.

In section 2, layers are declared with nodes and lowest level switch layers first. Although the symbols `l0`, `l1`, `l2`, and `l3` are arbitrary names for layers, and layer connection is determined separately by connect statements (section 3), their ordering determines ID assignment. The first node of the first layer declaration, n1 in this case, is assigned ID 0 and ID incremented for every node encountered in subsequent layer declaration statements. Switch IDs start at 200000, maintaining a separate counter, with ID incremented and assigned according to the same method. In this example, nodes are assigned ID 0 - 63 and switches 200000 - 200047.

The first connect statement in section 3 connects the nodes layer `l0` to switch layer 1 `l1` using 3ns for in/out channel delay. In effect this connects the NICs bundled with the node to switches in the first layer. The second and third connect statements are similar but use different values for channel delay and make use of a mapping vector. The first connect statement, using the mapping vector `[1,0,2]`, directs the parser to use the first base-four ID digit of source switch as the destination port, the first digit of the destination switch's base-four ID is equal to the source port number and the second digit of the destination equals the second ID of the source.[1] This operation is applied across all components of the source layer.

Finally, section 4 directs output to two files, `runtime.dml` and `model.dml`. The first output statement `model.dml{}`, because no values are included between the brackets, copies the contents of `components/main`, the default top-level output template, to `model.dml`, interpolating parser values set for `$nodes`, `$switches`, and `$connections`. This completes the `model.dml`. The second statement `runtime.dml{...}` simply outputs its contents to `runtime.dml`.

```
# section 1: parameter assignment
n:MESSAGE_DELAY{11000}
n:PACKET_DELAY{120}
n:PACKET_SIZE{320}
n:BUS_BANDWIDTH{950}
```

```
n:NET_BANDWIDTH{400}
n:ACK_BYTES{999999}
n:channel_delay{1.05263}

n1:MESSAGES{
        MESSAGE [
          TIME   0
          TARGET "1"
          SIZE   384
        ]
        MESSAGE [
          TIME   28000
          TARGET "1"
          SIZE   704
        ]
        MESSAGE [
          TIME   84000
          TARGET "1"
          SIZE   1024
        ]
        MESSAGE [
          TIME   168000
          TARGET "1"
          SIZE   1344
        ]
}

# section 2: layer declarations
l0=n1,n[63]
l1=s[16]
l2=s[16]
l3=s[16]

# section 3: connections
connect l0,l1 [] [3,3]
connect l1,l2 [1,0,2] [1,1]
connect l2,l3 [2,1,0] [10,10]

# section 4: output
model.dml{}
runtime.dml{EXECUTABLE      "parsim3"
MODEL           "model.dml"
MACHINE         "machine.dml"
STARTTIME       0
ENDTIME         4.44e+06
ENVIRONMENT [
  WOLVERINE     "1"
  LOG_STYLE     "IMMEDIATE"
  LOG_MASK      "ERROR WARNING INFO"
  DATA_STYLE    "AT WRAPUP"
  DATA_MASK     "INFO"
  EPILOG_STYLE  "IMMEDIATE"
  EPILOG_FILE   "wolverine.elg"
  WORKLOAD      "TestWorkload"
  NETWORK       "MFQuadrics"
  FIRST_NODE    0
  FIRST_NIC     100000
  FIRST_SWITCH  200000
  MPI_COMM_SIZE "64"
]
DATAFILE        "output"
}
```

5 Large-scale Representations

The example below is an idealized representation of a 4096 node machine. The network is arranged in a fat-tree of six layers:

```
l0=n[4096]
l1=s[1024]
l2=s[1024]
l3=s[1024]
l4=s[1024]
l5=s[1024]
l6=s[1024]

connect l0,l1
connect l1,l2 [1, 0, 2, 3, 4, 5]
connect l2,l3 [2, 0, 1, 3, 4, 5]
connect l3,l4 [3, 4, 0, 1, 2, 5]
connect l4,l5 [1, 0, 2, 3, 4, 5]
connect l5,l6 [5, 1, 2, 3, 4, 0]

model.dml{}
```

This representation produces an output model of approximately 6.1MB. Component parameters are applied uniformly, producing rather redundant DML code, but the resulting interconnection network is complex. A network diagram is useful in providing a quick method of visual verification in complex network topologies. After successful parsing of the model description file, internal structures used for DML translation are used to render a simple network diagram.

References

[1] The *à la carte* Project. *An Approach to Extreme-Scale Simulation of Novel Architectures.* Los Alamos National Laboratory, October 2001.

[2] The *à la carte* Project. *Design and Implementation of Low- and Medium-Fidelity Network Simulations of a 30-TeraOPS System.* Los Alamos National Laboratory, April 2002.

[3] Brian W. Bush. *Mathematical Description of the QSC Layout.* Los Alamos National Laboratory, April 2002.

[4] Jason Liu and David M. Nicol. *Dartmouth Scalable Simulation Framework User's Manual.* Dartmouth College, Department of Computer Science, August 2001.

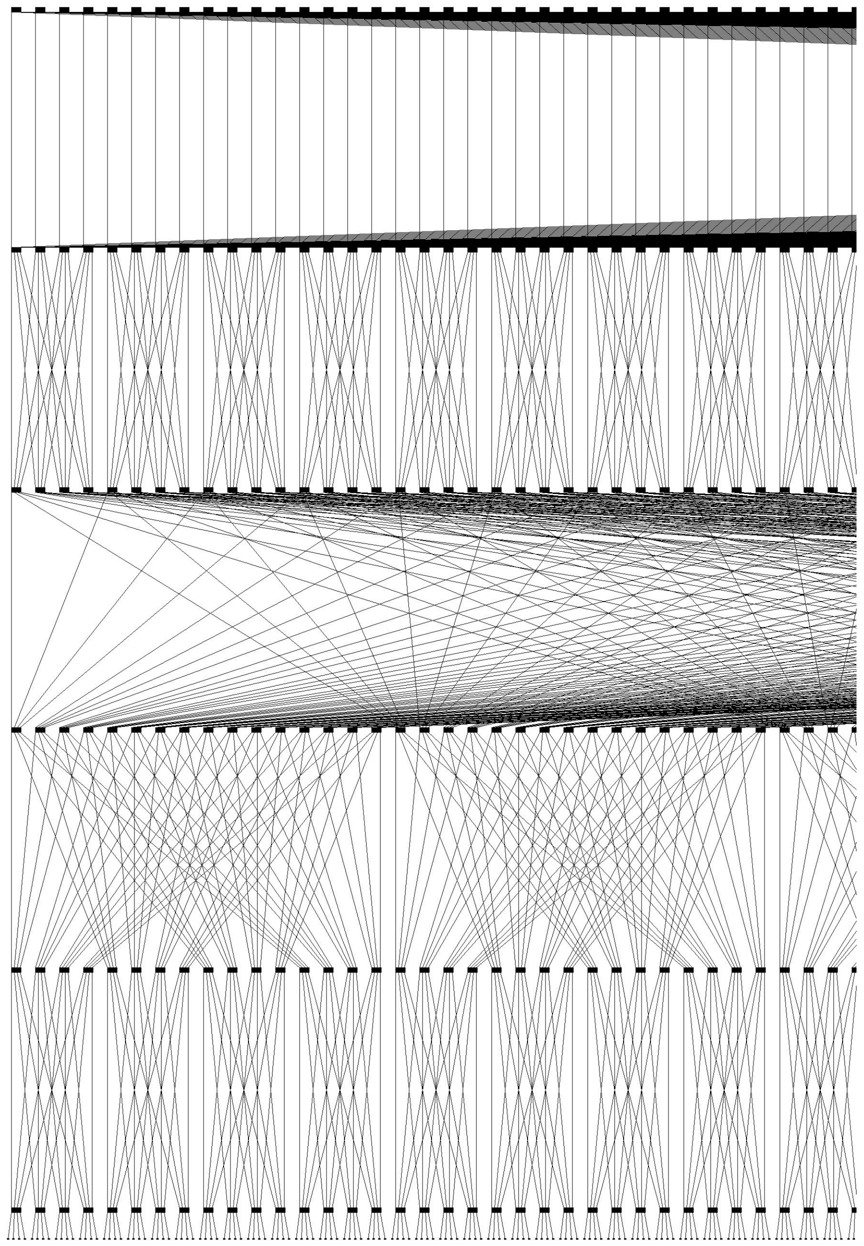

Figure 1: A simple network diagram showing the left 1/5 portion of an idealized 4096 node architecture whose network is a arranged in a six-layer quarternary tree. Nodes are pictured in the bottom layer.